Institut supérieur des études technologiques de Sousse
Mastère : Génie logiciel et développement rapide d'application
Enseignant: Jaafar CHAAOURI

Chapitre I : Introduction au Deep learning

Introduction

Intelligence artificielle: domaine de l'informatique qui vise à amener les ordinateurs à atteindre une intelligence de style humain. Il existe de nombreuses approches pour atteindre cet objectif, notamment l'apprentissage automatique (**machine learning**) et l'apprentissage en profondeur (**deep learning**).

- Machine Learning: Les systèmes de ML sont entrainés pour effectuer une tâche particulière plutôt que de les programmer explicitement. Ainsi les modèles obtenus apprennent comment combiner des entrées pour formuler des prédictions efficaces sur des données qui n'ont encore jamais été observées.
- Réseau de Neurones: Une construction de Machine Learning inspirée du réseau de neurones (cellules nerveuses) du cerveau biologique. Les réseaux de neurones sont un élément fondamental du Deep learning.
- Deep Learning: Un sous-domaine du Machine Learning qui utilise des réseaux de neurones multicouches.

La terminologie de base en matière de Machine Learning.

Étiquettes

Une étiquette est le résultat de la prédiction; la variable y dans une régression linéaire simple. Il peut s'agir de l'espèce animale représentée sur une photo, de la signification d'un extrait audio ou de toute autre chose.

Caractéristiques

Une caractéristique est une variable d'entrée ; la variable x dans une régression linéaire simple. Un projet de Machine Learning simple peut utiliser une seule caractéristique, tandis qu'un projet plus sophistiqué en utilisera plusieurs millions, spécifiées sous la forme :

> $x_1, x_2, .. x_n$

Dans l'exemple du détecteur de spam, les caractéristiques peuvent inclure les éléments suivants :

- les mots dans le corps de l'e-mail ;
- l'adresse de l'expéditeur ;
- l'heure à laquelle l'e-mail a été envoyé ;
- l'e-mail contient l'expression "Une astuce étrange".

Exemples

Un exemple est une instance de donnée particulière, x. (x est mis en gras pour indiquer qu'il s'agit d'un vecteur.) Les exemples se répartissent dans deux catégories :

- Exemples étiquetés
- Exemples sans étiquette

Un exemple étiqueté comprend une ou plusieurs caractéristiques et l'étiquette. Par exemple :

labeled examples: { $features, label$ }:(x, y)

On utilise des exemples étiquetés pour **entraîner** le modèle. Dans l'exemple du détecteur de spam, les exemples étiquetés désignent les e-mails que les utilisateurs ont explicitement marqués comme "spam" ou "non-spam".

Un exemple sans étiquette contient des caractéristiques, mais pas d'étiquette. Par exemple :

unlabeled examples: $\{features, ?\}$: $(x, ?)$

Une fois le modèle entraîné avec des exemples étiquetés, on l'utilise pour prédire l'étiquette sur des exemples qui en sont dépourvus. Dans l'exemple du détecteur de spam, les exemples sans étiquette sont des nouveaux e-mails qui n'ont pas encore été étiquetés manuellement.

Modèles :

Un modèle définit la relation entre les caractéristiques et l'étiquette. Par exemple, un modèle de détection de spam peut associer étroitement certaines caractéristiques à étiquette "spam". les deux phases de la durée de vie d'un modèle :

- **L'apprentissage** consiste à créer ou à **entraîner** le modèle. En d'autres termes, vous présentez au modèle des exemples étiquetés, et vous lui permettez d'apprendre progressivement les relations entre les caractéristiques et l'étiquette.
- **L'inférence** consiste à appliquer le modèle entraîné à des exemples sans étiquette. En d'autres termes, vous utilisez le modèle entraîné pour faire des prédictions efficaces (y').

Différence entre régression et classification

Les modèles de **régression** prédisent des valeurs continues. Ils formulent, par exemple, des prédictions qui répondent à des questions telles que :

- Quelle est la valeur d'un logement en Tunisie ?
- Quelle est la probabilité qu'un utilisateur clique sur cette annonce ?

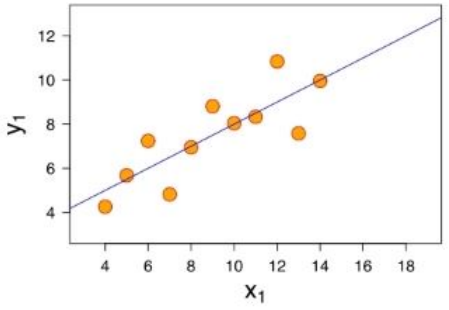
Les modèles de **classification** prédisent des valeurs discrètes. Ils formulent, par exemple, des prédictions qui répondent à des questions telles que les suivantes :

- Un e-mail donné est-il considéré comme du spam ou non ?
- Cette image représente-t-elle un chien, un chat ou un hamster ?

Régression linéaire

La régression linéaire est un algorithme qui va trouver une droite qui se rapproche le plus possible d'un ensemble de points. Les points représentent les données d'entraînement (Training Set).

Schématiquement, on veut un résultat comme celui là :



Les points en orange sont les données d'entrée (input data). Ils sont représentés par le couple (x_i, y_i). Les valeurs x_i sont les variables prédictives, et y_i est la valeur observée (le prix d'une maison par exemple). On cherche à trouver une droite : $F(x) = \alpha * x + \beta$ tel que, quelque soit x_i, on veut que $F(x_i) \approx y_i$.

En d'autres termes, **on veut une droite qui soit le plus proche possible de tous les points de nos données d'apprentissage**.

Format des données :

Les données d'apprentissage sont au format CSV. Les données sont séparés par des virgules. La première colonne représente la population d'une ville et la deuxième colonne indique le profit d'un camion ambulant dans cette ville.Une valeur négative indique une perte.Le nombre d'enregistrements de nos données d'entrées est 97.

NB. Le fichier est disponible avec le support de cours.

Pour résoudre ce problème, on va prédire le profit (la variable Y) en fonction de la taille de la population (la variable prédictive X)

Chargement des diverses librairies utiles pour ce notebook

```
# chargement de bibliothéques
"""
instruction specifique pour utiliser matplotlib dans un notebook
quand on utilise les notebooks Jupyter pour utiliser Matplotlib
"""
%matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
```

Chargement du jeu de données

Tout d'abord, il faudra lire et charger les données contenues dans le fichier CSV. Python propose via sa librairie Pandas des classes et fonctions pour lire divers formats de fichiers dont le CSV.

```
df = pd.read_csv("univariate_linear_regression_dataset.csv")
```

La fonction read_csv(), renvoie un DataFrame. Il s'agit d'un tableau de deux dimensions contenant, respectivement, la taille de population et les profits effectués. Pour pouvoir utiliser les librairies de régression de Python, il faudra séparer les deux colonnes dans deux variables Python.

```
X =  df.iloc[:,0] #selection de la première colonne de notre dataset (indice 0)
Y =  df.iloc[:,1] #selection de la première colonne de notre dataset (indice 1)
```

Les variables X et Y sont maintenant de simples tableaux contenant 97 éléments.

Note :

- La fonction len() permet d'obtenir la taille d'un tableau
- La fonction iloc permet de récupérer une donnée par sa position
- iloc[0:len(df),0] permettra de récupérer toutes les données de la ligne 0 à la ligne 97 (qui est len(df)) se trouvant à la colonne d'indice 0

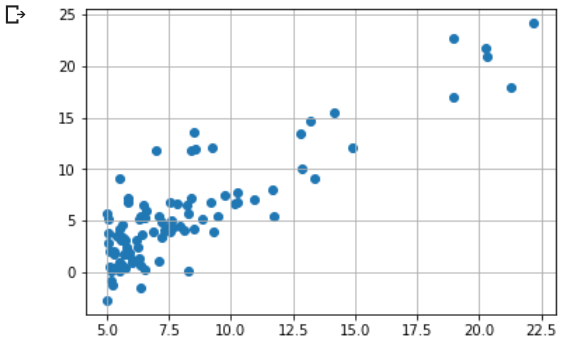
Visualisation du jeu de données

Avant de modéliser un problème de Machine Learning, il est souvent utile de comprendre les données. Pour y arriver, on peut les visualiser dans des graphes pour comprendre leur dispersion, déduire les corrélations entre les variables prédictives etc…

Parfois, il est impossible de visualiser les données car le nombre de variables prédictives est trop important. Ce n'est pas le cas ici, on ne dispose que de deux variables : la population et les profits.

Nous pouvons utiliser un graphe de type nuage de points (Scatter plot) pour visualiser les données :

```
axes = plt.axes()
axes.grid()
plt.scatter(X,Y)
plt.show()
```



On voit clairement qu'il y a une corrélation linéaire entre les variables. Et que plus la taille de la population augmente, plus le profit en fait de même.

Entrainement d'un modèle de régression linéaire

Note :

On peut utiliser le module librairie SciPy (Scientific Python) pour implémenter une régression linéaire. Le sous package stats propose la fonction linregress qui calcul une régression à partir d'un jeu de donnée d'entrainement

```
slope, intercept, r_value, p_value, std_err = stats.linregress(X, Y)
```

Modèle obtenu

Le fonction de prédiction pour une régression linéaire univariée est comme suit :

$$H(x) = intercept + slope * x$$

avec :

- $slope$: représente la "pente" de la line de prédiction
- $intercept$: représente le point d'intersection avec l'axe des ordonnées

Les coefficients de notre fonction de prédiction ont déjà été calculé et valent :

```
slope, intercept
```

```
(1.2135472539083585, -4.211504005424089)
```

Ainsi notre fonction $H(X)$ se décrit comme suit :

$$H(X) = -4.211504005424089 + 1.2135472539083585 * X$$

note :

- les valeurs de $slope$ et $intercept$ peuvent variées un peu en fonction des valeurs calculées par la fonction linregress et la précision de nombres flottants.

Ecriture de la fonction de prédiction avec Python

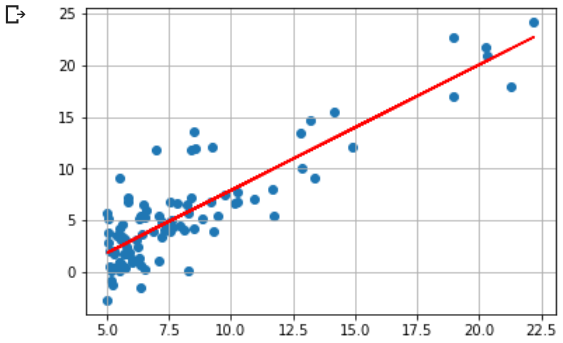
Vu qu'on dispose de notre fonction des coefficients de notre fonction de prédiction, on peut l'écrire en python.

```
# définition de quatre observations
def predict(x):
    return slope * x + intercept
```

Dessiner la fonction de prédiction

On peut utiliser la fonction de prédiction qu'on vient de définir pour avoir la valeur prédite par la fonction hypothèse pour chacune des observations de notre jeu d'entrainement. Ainsi on pourra voir visuellement à comment la fonction de prédiction "approche" le jeu d'entrainement et qu'elle est par conséquent une bonne fonction de prédiction.

```
axes = plt.axes()
axes.grid()
plt.scatter(X,Y)
fitLine = predict(X)
plt.plot(X, fitLine, c='r')
plt.show()
```



En effet, on voit bien que la ligne rouge, approche le plus possible tous les points du jeu de données.

Prédiction d'une nouvelle observation

On voit que pour pour la valeur x = 22.5, la valeur de y pour est environ 25. Utilisons la fonction $predict$ pour trouver une estimation de $H(x = 22.5)$

```
predict(22.5)
```

```
23.093309207513975
```

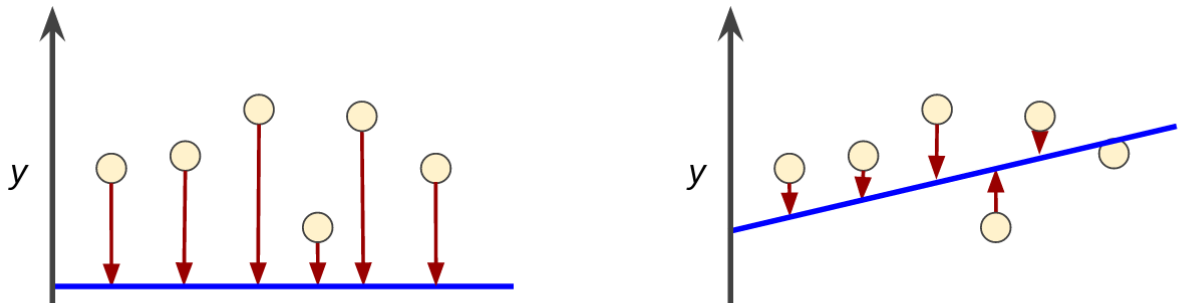
Assez proche ! 22.5 $\approx$ 23.09

Apprentissage et perte

Pour un modèle, l'**apprentissage** signifie déterminer les bonnes valeurs pour toutes les pondérations et le biais à partir d'exemples étiquetés. Dans l'apprentissage supervisé, un algorithme de Machine Learning crée un modèle en examinant de nombreux exemples, puis en tentant de trouver un modèle qui minimise la perte. Ce processus est appelé **minimisation du risque empirique**.

La perte correspond à la pénalité pour une mauvaise prédiction. Autrement dit, la perte est un nombre qui indique la médiocrité de la prévision du modèle pour un exemple donné. Si la prédiction du modèle est parfaite, la perte est nulle. Sinon, la perte est supérieure à zéro. Le but de l'entraînement d'un modèle est de trouver un ensemble de pondérations et de biais pour lesquels la perte, en moyenne sur tous les exemples, est faible. Par exemple, la figure 3 présente à gauche un modèle dont la perte est élevée, et à droite un modèle dont la perte est faible. À noter concernant cette figure :

- Les flèches rouges représentent les pertes.
- La ligne bleue représente les prédictions



Notez que les flèches rouges dans le graphique de gauche sont plus longues que celles de l'autre graphique. Il est clair que la ligne bleue dans le modèle de droite correspond à un modèle prédictif plus performant que celui représenté dans le graphique de gauche.

Vous vous demandez peut-être s'il est possible de créer une fonction mathématique (de perte) capable d'agréger les pertes de manière significative.

Perte quadratique :

Les modèles de régression linéaire que nous examinerons ici utilisent une fonction de perte appelée perte quadratique (ou perte L2). Pour un seul exemple, la perte quadratique est :

```
= the square of the difference between the label and the prediction
= (observation - prediction(x))2
= (y - y')2
```

L'erreur quadratique moyenne (MSE) correspond à la perte quadratique moyenne pour chaque exemple. Pour calculer l'erreur MSE, il faut additionner toutes les pertes quadratiques de chaque exemple, puis diviser cette somme par le nombre d'exemples :

$$MSE = \frac{1}{N}\sum_{x,y}(y - prediction(x))^2$$

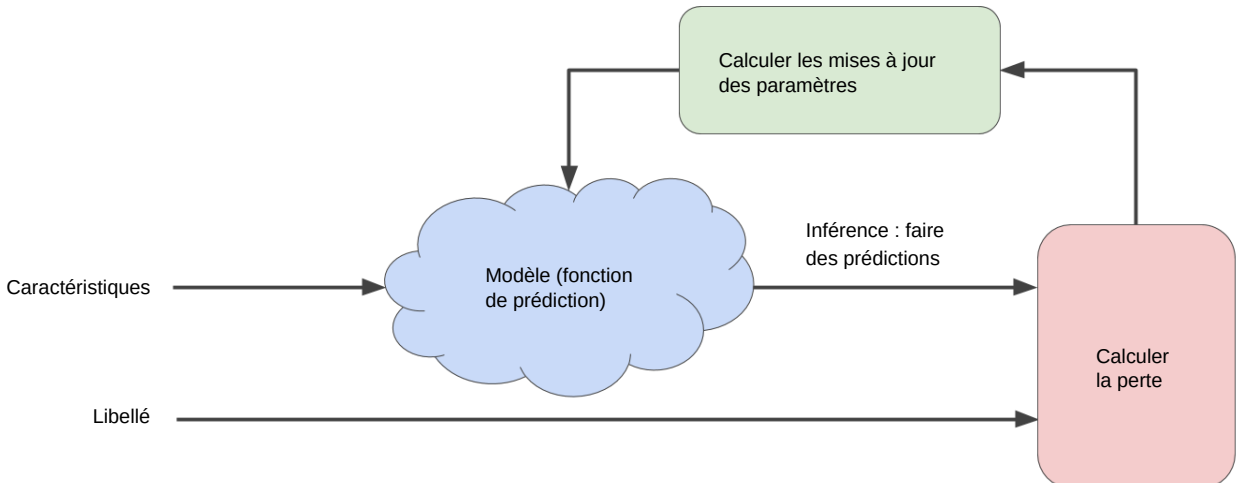
où :

- (x, y) est un exemple dans lequel :
- x est l'ensemble des caractéristiques (par exemple, température, âge et réussite de l'accouplement) que le modèle utilise pour réaliser des prédictions ;
- y est l'étiquette de l'exemple (par exemple, stridulations/minute).
- $prediction(x)$ est une fonction des pondérations et biais en combinaison avec l'ensemble des caractéristiques x.
- D est un ensemble de données contenant de nombreux exemples étiquetés, qui sont des paires (x, y).
- N est le nombre d'exemples dans D.

Bien que l'erreur MSE soit couramment utilisée dans le Machine Learning, ce n'est ni la seule fonction de perte pratique, ni la meilleure fonction de perte pour toutes les circonstances.

Réduction de la perte : une approche itérative

- Le module précédent a présenté le concept de perte. Dans ce module, vous découvrirez comment un modèle de Machine Learning peut minimiser la perte.
- Le schéma suivant illustre le processus itératif de tâtonnement utilisé par les algorithmes de Machine Learning pour l'entraînement d'un modèle.



Nous utiliserons cette même approche itérative pendant l'intégralité du cours d'initiation au Machine Learning, en détaillant diverses complications, en particulier celles qui se produisent à l'étape symbolisée par le nuage bleu. Les stratégies itératives sont largement utilisées pour le Machine Learning, car elles s'adaptent parfaitement aux ensembles de données de grande taille.

Le "modèle" accepte une caractéristique ou plus en entrée, et permet d'obtenir une prédiction (y') en sortie. Simplifions en imaginant le cas d'un modèle acceptant une caractéristique et permettant d'obtenir une prédiction.

> $$y' = b + w_1 * x_1$$

Quelles valeurs initiales devons-nous attribuer à w_1 et à b ? Dans les problèmes de régression linéaire, il s'avère que les valeurs de départ n'ont aucune importance. Nous pourrions les déterminer de façon aléatoire, mais nous utiliserons plutôt ici les valeurs arbitraires suivantes :

> - $b = 0$
> - $w_1 = 0$

Imaginons que la première valeur de caractéristique soit égale à 10. En utilisant cette valeur de caractéristique dans la fonction de prédiction, on obtient le résultat suivant :

> $$y' = 0 + 0(10)$$
>
> $$y' = 0$$

La case "Fonction de calcul de la perte" dans le schéma correspond à la fonction de perte utilisée par le modèle. Imaginons que nous utilisions la fonction de perte quadratique. La fonction de perte accepte deux valeurs d'entrée :

- y' : la prédiction du modèle pour les caractéristiques x
- y : l'étiquette correcte correspondant aux caractéristiques x.

Nous avons atteint l'étape "Calcul de la mise à jour des paramètres" dans le schéma. C'est à cette étape que le système de Machine Learning examine la valeur de la fonction de perte et génère de nouvelles valeurs pour et b et w_1.

Pour le moment, partez du principe que cette mystérieuse case verte détermine de nouvelles valeurs, puis que le système de Machine Learning compare à nouveau toutes ces caractéristiques aux étiquettes existantes, attribuant une nouvelle valeur à la fonction de perte, laquelle génère de nouvelles valeurs de paramètres.

Ainsi, l'apprentissage poursuit ses itérations jusqu'à ce que l'algorithme identifie les paramètres du modèle présentant la perte la plus basse possible. L'itération se poursuit généralement jusqu'à ce que la perte globale cesse d'évoluer ou n'évolue plus que très lentement. À ce stade, on déclare généralement que le modèle a convergé.

L'apprentissage d'un modèle de Machine Learning s'effectue à partir d'une valeur initiale de biais et de pondération aléatoire ajustée de façon itérative jusqu'à obtenir les valeurs de pondération et de biais présentant la perte la moins élevée possible.

Réduction de la perte : la descente de gradient

Le schéma représentant l'approche itérative (figure 1) contenait une étape très allusive intitulée "Calculer les mises à jour des paramètres". Remplaçons ce tour de algorithmique par des informations plus concrètes.

Imaginons que nous disposions du temps et des ressources informatiques nécessaires pour calculer la perte correspondant à toutes les valeurs possibles de w_1. Pour les types de problèmes de régression examinés jusqu'ici, le tracé représentant la perte rapportée à sera toujours convexe. En d'autres termes, ce tracé aura toujours la forme d'un bol, comme dans la figure suivante :

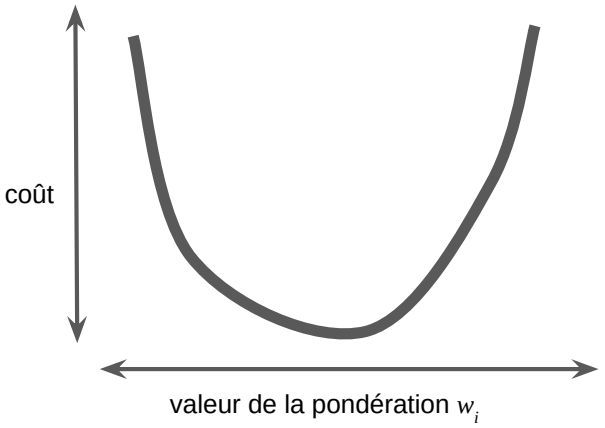


Figure 2 : Les problèmes de régression se traduisent par des tracés perte/pondération convexes.

- Les problèmes convexes possèdent un minimum unique : leurs courbes ne présentent qu'un point dont la pente est exactement égale à 0. Ce minimum désigne le point de convergence de la fonction de perte.
- Il serait inefficace de déterminer le point de convergence en calculant la fonction de perte pour chaque valeur imaginable de w_1 pour l'intégralité de l'ensemble de données. Il existe pour cela un meilleur outil, très utilisé pour le Machine Learning, appelé **la descente de gradient.**
- La première étape de l'application de la descente de gradient consiste à choisir une valeur de départ pour w_1. Cette valeur de départ importe peu. De nombreux algorithmes attribuent à w_1 la valeur 0 ou une valeur aléatoire. Comme le montre la figure suivante, nous avons choisi un point légèrement supérieur à 0.

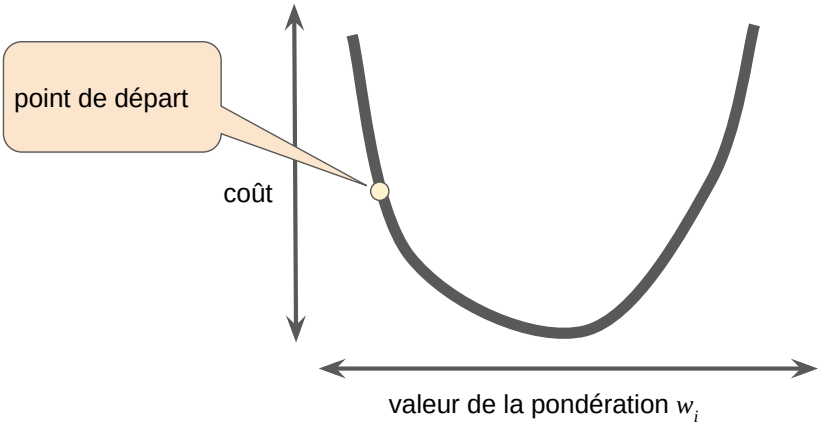


Figure 3 : Point de départ d'une descente de gradient

L'algorithme de descente de gradient calcule ensuite le gradient de la courbe de perte au point de départ. En bref, un **gradient** est un vecteur de dérivées partielles indiquant la direction à suivre pour atteindre le minimum recherché. Comme vous pouvez le constater, le gradient de perte est équivalent à la dérivée pour chaque valeur de pondération (comme illustré en figure 3).

Un gradient étant un vecteur, il présente les deux caractéristiques suivantes :

- une direction
- une magnitude

Le gradient indique toujours la direction de la croissance maximale de la fonction de perte. L'algorithme de descente de gradient fait un pas dans le sens inverse afin de réduire la perte aussi rapidement que possible.

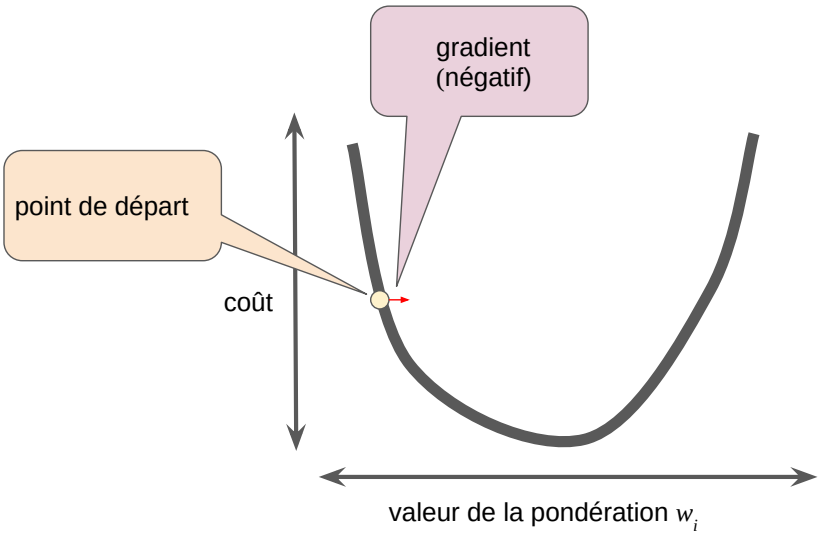


Figure 4 : La descente de gradient utilise des gradients négatifs.

Pour déterminer le point suivant dans la courbe de la fonction de perte, l'algorithme de descente de gradient ajoute une fraction de la magnitude du gradient au point de départ, comme illustré dans la figure suivante :

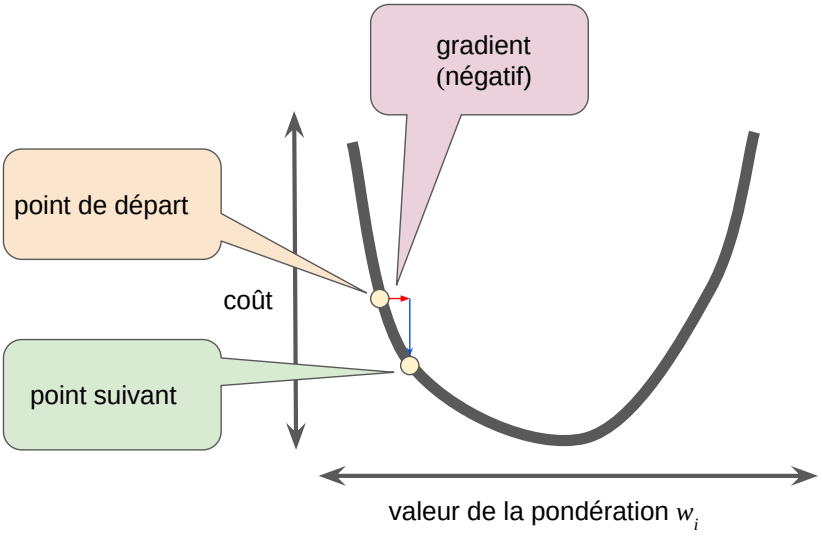


Figure 5 : Un pas de gradient nous positionne au point suivant de la courbe de perte.

La descente de gradient répète alors ce processus, se rapprochant ainsi progressivement du minimum.

Réduction de la perte : le taux d'apprentissage

Comme nous l'avons vu, un vecteur de gradient comporte à la fois une direction et une magnitude. Les algorithmes de descente de gradient multiplient généralement le gradient par une valeur scalaire appelée **taux d'apprentissage** (ou parfois pas d'apprentissage) pour déterminer le point suivant. Par exemple, si la magnitude du gradient est de 2,5 et que le taux d'apprentissage est de 0,01, alors l'algorithme de descente de gradient sélectionnera le point suivant situé à une distance de 0,025 du point précédent.

Les **hyperparamètres** sont les variables pouvant être ajustées par les programmeurs dans les algorithmes de Machine Learning. La plupart des programmeurs spécialisés dans le Machine Learning consacrent une bonne partie de leur temps à ajuster le taux d'apprentissage. Si vous

sélectionnez un taux d'apprentissage trop bas, le temps requis pour l'apprentissage sera trop long.

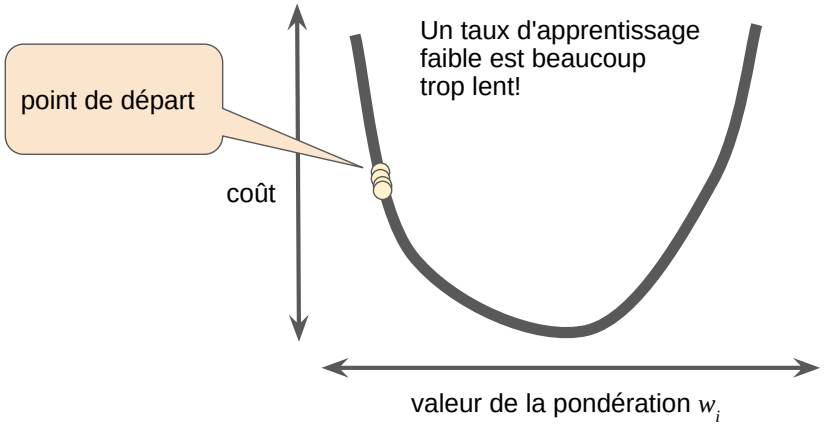


Figure 6 : Taux d'apprentissage trop bas

Dans le cas contraire, si vous fixez un taux d'apprentissage trop élevé, le point suivant rebondira frénétiquement de part et d'autre du minimum recherché, à la manière d'une expérience de mécanique quantique hors de contrôle :

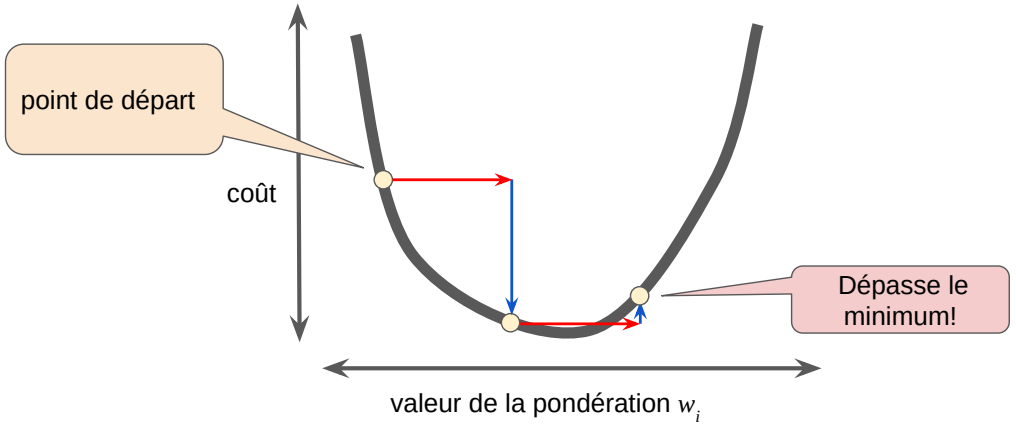


Figure 7 : Taux d'apprentissage trop élevé

Il existe un taux d'apprentissage idéal pour chaque problème de régression. Cette valeur dépend de la courbure de la fonction de perte. Si vous savez que le gradient de la fonction de perte est faible, vous pouvez en toute sécurité essayer un taux d'apprentissage plus élevé, ce qui compense le gradient faible et entraîne un pas d'apprentissage plus grand.

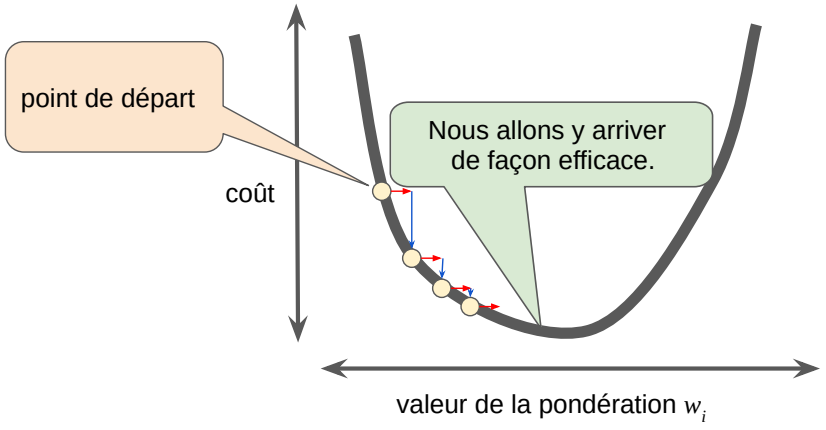


Figure 8 : Taux d'apprentissage adéquat

Réduction de la perte : la descente de gradient stochastique

Lors d'une descente de gradient, un **lot** représente le nombre total d'exemples utilisés pour calculer le gradient à chaque itération. Jusqu'à présent, nous avons procédé comme si le lot correspondait à l'intégralité de l'ensemble de données. À grande échelle, les ensembles de données contiennent souvent des milliards, voire des centaines de milliards d'exemples. En outre, les ensembles de données de Google comportent souvent un très grand nombre de caractéristiques. Un lot peut donc être gigantesque. Lorsque le lot est très important, la durée des calculs pour une simple itération peut être particulièrement longue.

Un ensemble de données important qui comporte des exemples échantillonnés de façon aléatoire contient généralement des données redondantes. De fait, plus la taille d'un lot augmente, plus la probabilité de redondance sera élevée. Un certain niveau de redondance peut être utile pour atténuer les effets du bruit dans les gradients, mais les lots gigantesques présentent rarement une valeur prédictive supérieure à celle des lots de grande taille.

Et s'il était possible d'obtenir le gradient souhaité, en moyenne, avec un nombre de calculs nettement inférieur ? Le choix d'exemples aléatoires dans notre ensemble de données nous permet d'estimer une moyenne importante à partir d'une moyenne bien plus modeste (en acceptant une certaine quantité de bruit). La descente de gradient stochastique (SGD) constitue une application radicale de ce principe, car elle n'utilise qu'un exemple (un lot dont la taille est 1) par itération. Si le nombre d'itérations est assez important, la SGD fonctionne, tout en générant beaucoup de bruit. Le terme "stochastique" signifie que l'exemple constituant chaque lot est sélectionné de façon aléatoire.

La descente de gradient stochastique (SGD) par mini-lots (SGD par mini-lots) offre un compromis entre l'itération des lots entiers et la SGD. Un mini-lot comprend généralement entre 10 et 1 000 exemples sélectionnés aléatoirement. La SGD par mini-lots limite la quantité de bruit propre aux SGD tout en restant plus efficace que le traitement de lots entiers.

Nous avons appliqué la descente de gradient à une caractéristique unique. Naturellement, la descente de gradient fonctionne également sur des ensembles de caractéristiques comportant des caractéristiques multiples.

Les réseaux de neurones :

Le problème de classification suivant était non linéaire :

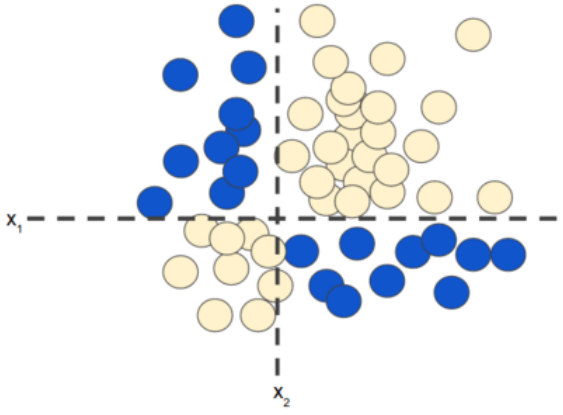


Figure 1 : Problème de classification non linéaire.

"Non linéaire" signifie qu'il n'est pas possible de prédire avec exactitude une étiquette avec un modèle de la forme

> $b + w_1 x_1 + w_2 x_2$

Autrement dit, la "surface de décision" n'est pas une droite.

Voyons à présent l'ensemble de données suivant :

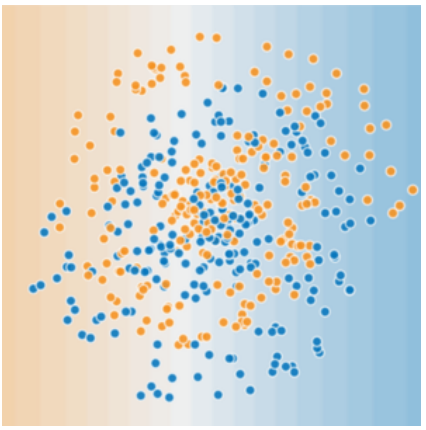


Figure 2 : Problème de classification non linéaire plus complexe.

L'ensemble de données illustré à la figure 2 ne peut pas être résolu avec un modèle linéaire.

Pour voir comment les réseaux de neurones peuvent aider à résoudre les problèmes non linéaires, commençons par représenter un modèle linéaire à l'aide d'un graphique :

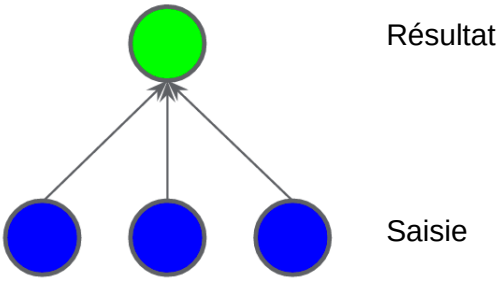


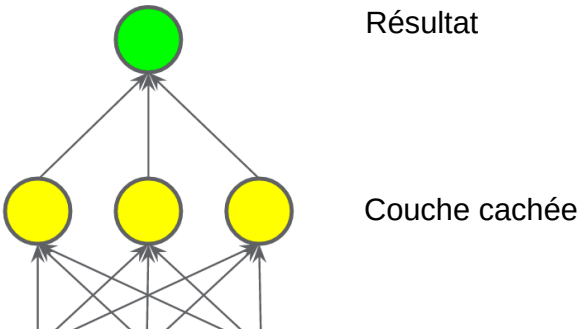
Figure 3 : Représentation graphique d'un modèle linéaire.

Chaque cercle bleu représente une caractéristique d'entrée, et le cercle vert représente la somme pondérée des entrées.

Comment pouvons-nous modifier ce modèle afin d'améliorer sa capacité à traiter les problèmes non linéaires ?

Couches cachées

Dans le modèle représenté par le graphique suivant, nous avons ajouté une "couche cachée" de valeurs intermédiaires. Chaque nœud jaune de la couche cachée est une somme pondérée des valeurs des nœuds d'entrée bleus. La sortie est la somme pondérée des nœuds jaunes.



Ce modèle est-il linéaire ? Oui. La sortie est une combinaison linéaire des entrées.

Dans le modèle représenté par le graphique suivant, nous avons ajouté une deuxième couche cachée de sommes pondérées.

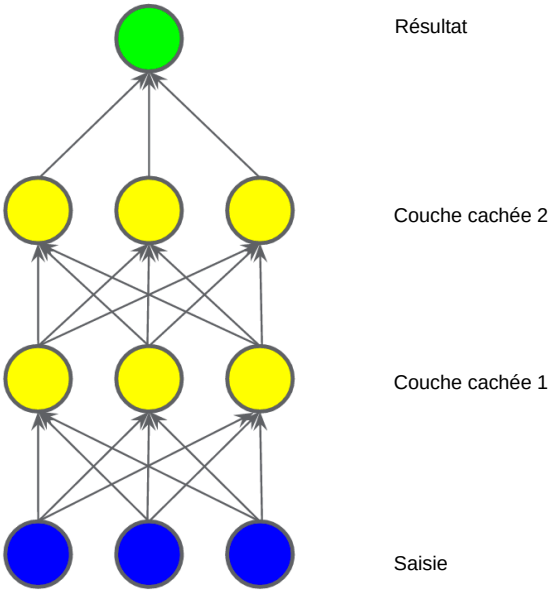
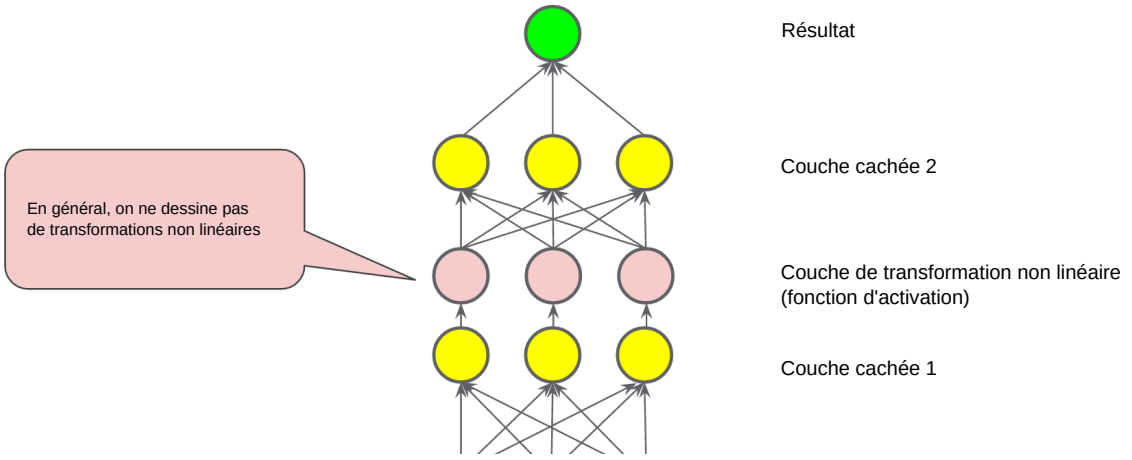


Figure 5 : Représentation graphique d'un modèle à trois couches.

Ce modèle est-il toujours linéaire ? Oui. Lorsque la sortie est exprimée comme une fonction de l'entrée, puis simplifiée, vous obtenez simplement une autre somme pondérée des entrées. Cette somme ne modélisera pas efficacement le problème non linéaire de la figure 2.

Fonctions d'activation



Maintenant que nous avons ajouté une fonction d'activation, l'impact de l'ajout de couches est plus important. En empilant des non-linéarités, nous pouvons modéliser des relations très complexes entre les entrées et les sorties prévues. Pour résumer, chaque couche apprend une fonction plus complexe, d'un niveau supérieur, des entrées brutes.

Figure 6 : Representation graphique d'un modele a trois couches avec fonction d'activation.

Fonctions d'activation courantes

La fonction d'activation sigmoïde suivante convertit la somme pondérée en une valeur comprise entre 0 et 1.

$$F(x) = \frac{1}{1+e^x}$$

Voici sa représentation graphique :

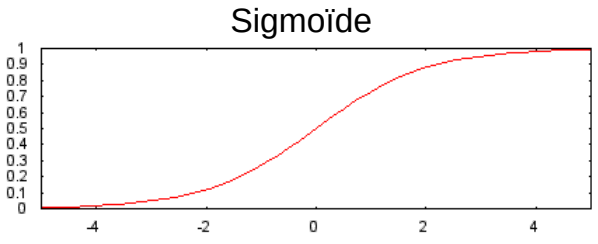


Figure 7 : Fonction d'activation sigmoïde.

La fonction d'activation d'unité de rectification linéaire (ou ReLU) est souvent un peu plus efficace qu'une fonction lisse de type sigmoïde, tout en étant bien plus simple à calculer.

$$F(x) = max(0, x)$$

La supériorité de la fonction ReLU repose sur des conclusions empiriques, sans doute du fait que la fonction ReLU présente une plage de réponse plus utile. La réponse de la fonction sigmoïde est rapidement défaillante de chaque côté.

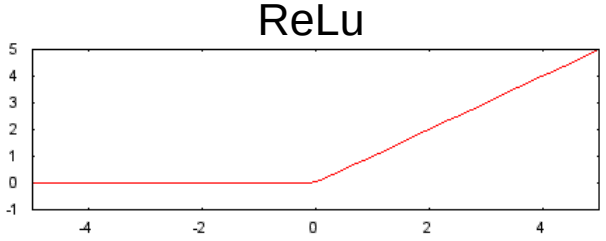


Figure 8 : Fonction d'activation ReLU.

En fait, toute fonction mathématique peut être utilisée comme fonction d'activation. Supposons que représente notre fonction d'activation (ReLU, sigmoïde ou autre). La valeur d'un nœud dans le réseau est alors donnée par la formule suivante :

> $\sigma(w.x + b)$

TensorFlow fournit une aide prête à l'emploi pour une large gamme de fonctions d'activation. Cela étant dit, nous recommandons quand même de commencer par la fonction ReLU.

Résumé Notre modèle possède à présent tous les composants standards de ce que l'on appelle généralement un "réseau de neurones" :

- Un ensemble de nœuds, semblables à des neurones, organisés en couche.
- Un ensemble de pondérations, représentant les connexions entre chaque couche du réseau de neurones et la couche inférieure. Cette couche inférieure peut être une autre couche de réseau de neurones ou un autre type de couche.
- Un ensemble de biais, un par nœud.
- Une fonction d'activation qui transforme la sortie de chaque nœud d'une couche. Différentes couches peuvent avoir différentes fonctions d'activation.

Entraîner les réseaux de neurones : bonnes pratiques

Cette section explique les problèmes possibles de la rétropropagation, ainsi que la méthode la plus utilisée pour régulariser un réseau de neurones.

- Problèmes possibles

La rétropropagation peut être à l'origine de plusieurs problèmes courants.

- Disparition des gradients

Les gradients des couches inférieures (qui sont les plus proches de l'entrée) peuvent devenir extrêmement petits. Dans les réseaux profonds, le calcul de ces gradients peut impliquer le produit de nombreux petits termes.

Lorsque les gradients se rapprochent de 0 pour les couches inférieures, ces dernières sont entraînées très lentement, voire pas du tout.

La fonction d'activation des unités ReLU permet d'empêcher la disparition des gradients.

- Explosion des gradients

Si les pondérations d'un réseau sont très importantes, les gradients des couches inférieures impliquent le produit de nombreux termes de grande taille. Dans ce cas, les gradients peuvent exploser. Autrement dit, ils sont trop grands pour que la convergence fonctionne.

La normalisation de lot, tout comme la réduction du taux d'apprentissage, peut empêcher l'explosion des gradients.

- Unités ReLU inactives

Lorsque la somme pondérée d'une unité ReLU descend en dessous de 0, l'unité peut se figer. Elle génère alors 0 activation et ne contribue donc pas à la sortie du réseau, tandis que les gradients ne peuvent plus y passer lors de la rétropropagation. En cas d'élimination d'une source de gradients, il se peut même que l'entrée effectuée dans l'unité ReLU ne puisse plus jamais changer suffisamment pour que la somme pondérée repasse au-dessus de 0.

La réduction du taux d'apprentissage peut empêcher les unités ReLU de devenir inactives.

- Régularisation par abandon

Une autre forme de régularisation, appelée abandon, est utile pour les réseaux de neurones. Cette méthode "abandonne" de manière aléatoire des activations d'unités dans un réseau pour un pas de gradient unique. Plus il y a d'abandons, plus la régularisation est poussée :

0.0 = pas de régularisation par abandon. 1.0 = abandon total (le modèle n'apprend rien). Les valeurs comprises entre 0.0 et 1.0 sont plus efficaces.

Réseaux de neurones à classes multiples : un contre tous

Un contre tous permet d'utiliser la classification binaire. Étant donné un problème de classification avec N solutions possibles, une solution un contre tous consiste en N classifieurs binaires distincts : un classifieur binaire pour chaque résultat possible. Au cours de l'apprentissage, le modèle parcourt une séquence de classifieurs binaires, formant chacun pour répondre à une question de classification distincte. Par exemple, prenons une image représentant un chien. Cinq reconnaissances différentes pourront être formées, dont quatre verront l'image comme un exemple négatif (pas un chien) et une verra l'image comme un exemple positif (un chien). Par exemple :

- Cette image représente-t-elle une pomme ? Non.
- Cette image représente-t-elle un ours ? Non.
- Cette image représente-t-elle des friandises ? Non.
- Cette image représente-t-elle un chien ? Oui.
- Cette image représente-t-elle un œuf ? Non.

Cette approche est relativement raisonnable lorsque nombre total de classes est réduit, mais devient de plus en plus inefficace à mesure que le nombre de classes augmente.

Nous pouvons créer un modèle un contre tous considérablement plus efficace avec un réseau de neurones profond, dans lequel chaque nœud de résultat représente une classe différente. La figure suivante suggère cette approche :

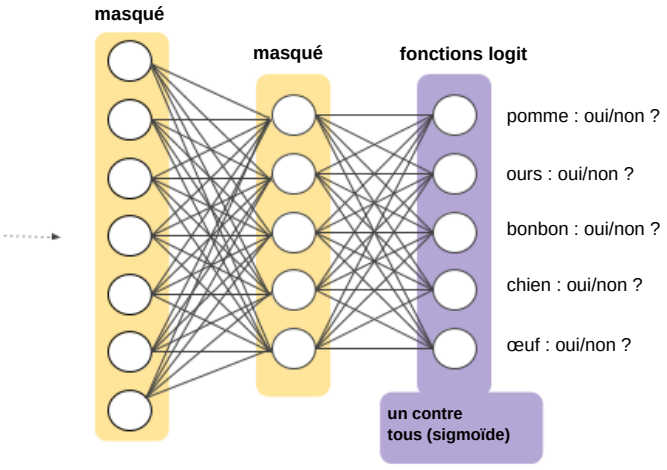


Figure 1 : Réseau de neurones un contre tous.

Réseaux de neurones à classes multiples : Softmax

Rappelez-vous que la régression logistique produit une décimale entre 0 et 1. Par exemple, un résultat de régression logistique de 0,8 pour un classificateur d'e-mails suggère que les chances qu'un e-mail soit indésirable sont de 80 %, et les chances qu'il ne soit pas indésirable de 20 %. De façon évidente, la somme des probabilités qu'un e-mail soit indésirable ou non est égale à 1.

Softmax étend cette idée à un monde à plusieurs classes. C'est-à-dire que Softmax attribue des probabilités décimales à chaque classe d'un problème à plusieurs classes. La somme de ces probabilités décimales doit être égale à 1. Cette contrainte supplémentaire permet de faire converger l'apprentissage plus rapidement qu'il ne le ferait autrement.

Par exemple, revenons à l'analyse de l'image de la Figure 1. Softmax pourra produire les probabilités suivantes qu'une image appartienne à une classe spécifique :

Classe	Probabilité
pomme	0,001
ours	0,04
friandises	0,008
chien	0,95
œuf	0,001

Softmax est mis en œuvre via une couche de réseau de neurones juste avant la couche du résultat. La couche Softmax doit comporter le même nombre de nœuds que la couche du résultat.

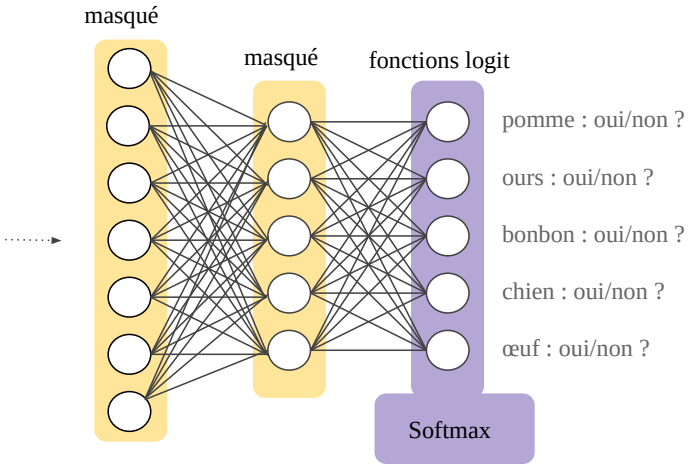


Figure 2 : Une couche Softmax dans un réseau de neurones.

Options de Softmax

Prenons les variantes de Softmax suivantes :

- Softmax complet est le Softmax dont nous avons parlé, c'est-à-dire le Softmax qui calcule une probabilité pour chaque classe possible.
- L'échantillonnage de candidats signifie que Softmax calcule une probabilité pour toutes les étiquettes positives, mais seulement pour un échantillon aléatoire d'étiquettes négatives. Par exemple, si nous souhaitons déterminer si une image d'entrée est un beagle ou un limier, il est inutile de fournir des probabilités pour chaque exemple "non chien".

Softmax complet est relativement économique lorsque le nombre de classes est petit, mais devient extrêmement coûteux lorsque le nombre de classes augmente. L'échantillonnage de candidats peut améliorer l'efficacité des problèmes qui comportent un grand nombre de classes.

Une étiquette ou plusieurs étiquettes

Softmax suppose que chaque exemple appartient exactement à une classe. Cependant, certains exemples peuvent appartenir simultanément à plusieurs classes. Pour ces exemples :

- Vous ne pouvez pas utiliser Softmax.
- Vous devez vous appuyer sur les régressions logistiques.

Par exemple, supposons que vos exemples sont des images contenant exactement un élément : un fruit. Softmax peut déterminer la probabilité que cet élément soit une poire, une orange, une pomme, etc. Si vos exemples sont des images contenant toute sorte de choses (des saladiers contenant plusieurs types de fruits), alors vous devrez utiliser plusieurs régressions logistiques.

Atelier de programmation

Basic classification: Classify images of clothing